Explorations on the Formal Frontier of Distributed System Design

Martin S. Feather *
USC / Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292, USA
Email: feather@isi.edu

Abstract

The study of distributed system design has tended to divide into two camps: either very formal methods that apply to severe abstractions of problems, or human-intuition-based methods that aid in forming the overall nature of a system design, but are often disconnected from the application of formal methods except as post-hoc verifications.

Herein I explore part of the boundary of formal design, seeking to apply formal techniques to aid in the derivation *of a distributed design. This is demonstrated on the rederivation of the train signalling protocol developed in the 19th century. It was this protocol that gave rise to the now-familiar concept of* semaphore. *However, a study of the train system shows that there is much more to its protocol than simply the instantiation of the semaphore concept.*

This exploration places on a more formal footing issues such as discovery of the need for information, and the origin of key invariants, without requiring that the solution be known in advance.

1 Introduction - the railroad semaphore

The familiar concept, and name, *semaphore,* as used in programming distributed systems, derives from the use of mechanical signals as part of the train management protocol developed in the 19th century. Work on programming language semaphores has abstracted away many of the details of the train problem in order to emerge with a conceptual building block for many of the more complex synchronization problems common to distributed computer systems. Fickas and Helm [10] found that the train management protocol itself is a rich and interesting example of distributed system design, whose complexity goes well beyond that of simply instantiating our modern-day abstract notion of semaphores. They used the problem as an example of design that they could rationally reconstruct, based on plausible reasoning that took into account safety and progress goals, interfaces, assignment of tasks to agents, and concerns for possible system failures. Their design is founded on a mixture of petri-nets and planning formalisms which, while suited to displaying the overall structure of the design, is somewhat removed from a more formal treatment of the program design involving semaphores. My purpose in this paper is to (at least partially) bridge the gap between the formal but highly abstracted world of semaphores and invariants, and the intuitive but informal design view of overall system design.

I begin by presenting the *solution* to the train-signalling protocol, section 2. This presentation is for expository purposes — the derivation that follows, section 3, does not assume that this solution is known. Related work and conclusions complete the paper.

2 Train management - the solution!

I now present the solution to the train-signalling protocol. This solution simplifies from some of the complexity of the actual train system, as will be explained shortly.

The essence of the problem is to ensure safe and expedient movement of *trains* along a portion of railroad track. Track is partitioned into what are called *blocks,* contiguous segments of track of some 5 to 15 miles length. Trains move along the track in one direction only. At the start of each block is a *signal.* Each sig-

*The author has been supported in part by Defense Advanced Research Projects Agency grant No. NCC-2-520, and in part by Rome Air Development Center contract No. F30602-85-C-0221 and F30602-89-C-0103. Views and conclusions contained in this document are those of the author and should not be interpreted as representing the official opinion or policy of DARPA, RADC, NSF, the U.S. Government, or any other person or agency connected with them.

0-8186-2880-4/92 $03.00 © 1992 IEEE

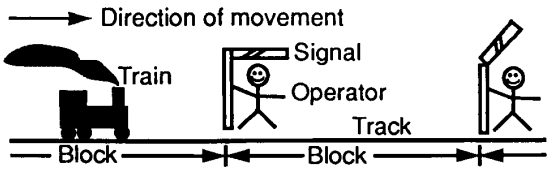

Figure 1: Configuration of train signalling system

nal has two possible settings, *'clear'*, indicating that a train in the preceding block is allowed to enter the signal's block, and *'stop'*, indicating that a train in the preceding block is not allowed to enter. Also at the start of each block is a station *operator* who: observes the entry of trains into that block, controls the setting of that block's signal, and communicates information to the station operator of the preceding block. Figure 2 illustrates this configuration.

The *safety condition* is that no two trains be in the same block at the same time - thus avoiding the danger of a collision. The *progress condition* is that trains eventually advance from block to block.

2.1 The protocol of the solution

From an initial configuration in which the safety condition is satisfied, that is, no two trains are in the same block, each occupied block's signal is set to 'stop' (to prohibit the entry of another train into that block), and in which each unoccupied block's signal is set to 'clear', the protocol functions as follows:

> At any time, a train in a block may move into the next block provided that the signal of that block is set to 'clear'.
>
> When a train enters a block, the station operator of that block observes the entry, and (eventually) performs the following two actions; first, sets the signal of that block to 'stop', then notifies the station operator of the preceding block of the movement of the train out of that preceding block.
>
> When a station operator is notified by the station operator of the next block (of the movement of a train from that first operator's block into that next block), he/she (eventually) performs the action of changing his/her block's signal to 'clear'.
>
> There is one complication in station operator behavior: a station operator may have observed entry of a train, but before acting to change the signal and notify the operator of the preceding block, is notified by the next block's station operator of train entry (i.e., the train has moved on to the next block, and the operator there has responded to its movement before the operator in this block has responded to its earlier movement). In this case, the operator can either serialize his/her pending actions by first setting his/her signal to 'stop', then notifying the station operator of the preceding block, and finally changing his/her signal back to 'clear', or may (safely) optimize this sequence of activity by leaving the signal in its 'clear' position, and simply notifying the station operator of the preceding block.

2.2 Observations on the solution

There are some subtle aspects to the above protocol:

- It is important that the station operator's response to entry of a train is to set his/her block's signal to 'stop' *before* (or simultaneously with) notifying the operator of the preceding block (the other way around leaves open the possibility that another train be allowed through before the operator gets around to changing the signal, which could lead to violation of the safety condition).

- When a station operator receives notification from the next operator and has yet to respond to the earlier movement of that train into his/her own block, it is important that he/she eventually both notifies the station operator of the preceding block, and leaves (or changes and changes back) his/her own block's signal to 'clear'. To act otherwise would lead to deadlock in which no further trains could get through, either because some preceding station operator has a signal set to 'stop' and is awaiting notification (which will now never materialize), or because the current block's signal becomes set to 'stop', and never gets changed back to 'clear' because no further notifications will ever come from ahead.

- Some degree of safety in the event of failure is built into this protocol – if a station operator is arbitrarily slow, even to the point of not taking any actions whatsoever, this can lead to deadlock, but not violation of the safety goal. Of course, performing incorrect actions can lead to violation of either goal (e.g., arbitrarily setting a signal to

'stop', moving a train past a signal in its 'stop' position).

- Finally, it is interesting to note that a block's signal being 'clear' is *not* equivalent to that block being empty; it does not even *imply* that the block is empty! In fact, the key to this protocol is the following invariant:

> For any two trains in blocks, there is always at least one block in front of the trailing train and at or before the leading train whose signal is set to 'stop'.

My goal is to *derive* this invariant from first principles.

2.3 Simplifications of the real-world problem

The real-world train semaphore protocol uses more complex signals, with *three* settings ('clear' and 'stop', the two settings outlined above, and 'caution', which warns a train to be prepared to stop), and uses signals placed not only at the start of a block, but also some distance ahead of the start of a block. Both of these elaborations permit better 'throughput', taking into account such real-world considerations such as inertia (trains cannot decelerate to a stop instantaneously, nor can they accelerate to full speed instantaneously, hence advance warning of the need, or lack of the need, to come to a halt can be useful). Presumably the placement and operation of these signals is justified by a continuous model of train movement. While the interplay between such a model and the simpler, discrete model that I outline is no doubt interesting, I have *not* studied it, and this paper does *not* attempt to deal with this.

3 (Partial) derivation of the train semaphore protocol

I now show how to use formal techniques to (at least partially) *derive* the train semaphore protocol from first principles.

3.1 Notation

I adopt a hybrid notation of Dijkstra's guarded commands [8] combined with global invariants. Guarded commands take the form

$$[]g \rightarrow S$$

where the *guard* g is a boolean condition and the *body* S is a sequential statement. Both guards and bodies may refer to global variables. Guards may also introduce local variables, to which their corresponding bodies may refer. I use set membership binding the value of a variable to the left of $[]$ to denote a set of guarded commands, one for each possible value of the variable, e.g.,

$$x \in X \,[]\, g \rightarrow S$$

denotes a set of guarded commands, one for each element of set X; the guard g and body S may use x to refer to that element.

3.2 Representing trains and blocks

I assume that the T is a set-valued global variable holding the set of trains present in the system. Likewise, B is a set-valued global variable holding the set of blocks that partition the track; for convenience, consecutive blocks are represented as consecutive integers; trains will always move only in the direction of increasing numbered blocks. The locations of trains are stored in the global array A; thus the expression $A[t]$ denotes the block in which train t is located. Simple behavior of trains is expressed as follows:

$$t \in T \,[]\, true \rightarrow A[t] := A[t] + 1 \tag{1}$$

That is, from the set of trains T, choose any train t, and move it forward from its current block (the value of $A[t]$) to the next block (the value of $A[t]+1$). The choice of which train to move is unconstrained, because the guard is *true* regardless of the value of t.

Notice that there's an implicit conditional here, namely that the train t be located in some block. For future convenience, I rewrite the above to make this explicit, thus:

$$t \in T \,[]\, (\exists b : A[t] = b) \rightarrow A[t] := A[t] + 1 \tag{2}$$

or, equivalently, using in the body the value for b established in the guard:

$$t \in T \,[]\, (\exists b : A[t] = b) \rightarrow A[t] := b + 1 \tag{3}$$

The safety condition we wish to impose is that no two trains be in the same block; this is expressed as an *invariant* thus:

$$\sim \exists t_1, t_2 : t_1 \neq t_2 \wedge A[t_1] = A[t_2] \tag{4}$$

3.3 Derivation method

I now apply a well-established method for *deriving* the necessary and sufficient conditions on train movement that ensure satisfaction of the invariant expressing the safety condition.

Given guarded commands (e.g., those expressing train movement, as in (3)) together with global invariants, there is a method for *deriving* preconditions (additional conjuncts on the guards) that ensure satisfaction of those invariants. This method works by determining, for each guarded command's body, the ***weakest precondition*** [8] that ensures the state resulting from executing that body will satisfy the invariant; together with the requirement that the invariant is true at the start of execution, this ensures the invariant holds throughout execution.

For the guarded commands (3) and invariant (4), this results in:

$$t \in T \,[]\, (\exists b : A[t] = b \wedge (\sim \exists t_1 : A[t_1] = b + 1)) \\ \rightarrow A[t] := b + 1 \qquad (5)$$

together with a requirement that the invariant (4) holds initially. In words, a train in some block b can move to the next block, $b + 1$, only if that next block is empty.

This is trivial for us to realize without calculation, hence the reader may regard this as a case of unnecessary formalization. However, as we will see later, more complex formulae benefit more from this formal calculation of the weakest preconditions.

The derivation method I am applying is that illustrated by Balzer in his transformational derivation of the 8 queens example [5] (his steps of 'Unfold constraint' – constraint means invariant – followed by 'move constraint test ahead of assertion'), and that applied by Andrews in his derivations of many synchronization problems [1, 2]. The earliest description I have been able to find of such an approach is that by Sintzoff [14]; a more complete treatment is presented by van Lamsweerde and Sintzoff [15].

The details of this derivation method are as follows:

- first, turn the invariant into a requirement on the initial state, and a *postcondition* on the body of every guarded command;
- then, for each such postcondition, calculate the weakest precondition of that postcondition with respect to that body, that is, the weakest condition that ensures execution of the body would lead to satisfaction of the statement;
- add this precondition as a conjunct of the guard;
- finally, *simplify* the precondition in its context, in particular, taking into account all the invariants of the program, *including* the one we are currently manipulating.

The introduction of postconditions on the above train example yields

$$t \in T \,[]\, (\exists b : A[t] = b) \rightarrow A[t] := b + 1 \ \textbf{post} \\ (\sim \exists t_1, t_2 : t_1 \neq t_2 \wedge A[t_1] = A[t_2]) \qquad (6)$$

(writing **post** to indicate that what follows is a postcondition of the preceding body). I.e., a train in some block b can move to the next block, $b + 1$, only if after it has done so there are no two trains in the same block.

Weakest precondition calculation yields:

$$t \in T \,[]\, (\exists b : A[t] = b \wedge (\sim \exists t_1 : A[t_1] = b + 1) \\ \wedge (\sim \exists t_1, t_2 : t_1 \neq t_2 \wedge A[t_1] = A[t_2])) \\ \rightarrow A[t] := b + 1 \qquad (7)$$

The conjunct $(\sim \exists t_1, t_2 : t_1 \neq t_2 \wedge A[t_1] = A[t_2])$ simplifies to true (because it is identical to the invariant we are ensuring), and the result of this is the guarded command (5) shown earlier.

3.4 Availability of information

In the physical world, a train in block b cannot necessarily see into the next block $b + 1$ to ascertain whether or not a train is present (recall that blocks are anywhere from 5 to 15 miles in length). Recasting the consequences of this into our formalism, this means that a train in block b cannot evaluate the predicate $(\sim \exists t_1 : A[t_1] = b + 1)$, i.e., the question 'Is block $b+1$ empty?' I believe that it to be straightforward to formally represent what a train 'knows' directly (e.g., a train t in block b knows directly: the value of $A[t]$, its own identity, and the configuration of blocks $[\ldots b-2, b-1, b, b+1, \ldots]$), and hence to calculate whether a train knows (i.e., knows directly, or can deduce from what it knows directly) the value of a predicate such as $(\sim \exists t_1 : A[t_1] = b + 1)$. I have not pursued such reasoning about knowledge; the reader is referred to, e.g., [11] for an overview of this kind of work.

Returning to the train's inability to evaluate the predicate that guards its action, this we see as a *deficiency* in the emerging design (following Fickas' use of terminology). Such deficiencies motivate the need for, and choices of, compensatory changes to either the goals (safety and progress conditions), the capabilities of the the agents (trains) already in the system,

and/or the introduction of new agents. It is a combination of the latter two that the actual train system employs — *signals* are introduced to communicate information to trains, station *operators* are introduced to control the settings of these signals, and train behavior is adjusted to react to signal settings. I now show how the signalling protocol can be semi-formally *derived*. This is in contrast to concocting the solution, and then verifying that it is correct.

First, begin with the problematic predicate that trains need to evaluate:

$$(\sim \exists t_1 : A[t_1] = b + 1) \tag{8}$$

The *invention* step is to propose the introduction of some sort of communication mechanism to pass this information to the train. Encapsulating this as a predicate (a truth-valued function) called OK, say, we must ensure the invariant:

$$\forall b \in B : OK[b] \equiv (\sim \exists t_1 : A[t_1] = b + 1) \tag{9}$$

and, using OK, may re-write train behavior (5) as:

$$t \in T \,[\!]\, (\exists b : A[t] = b \wedge OK[b+1]) \rightarrow A[t] := b + 1 \tag{10}$$

I.e., a train in some block b can move to the next block, $b+1$, only if $OK[b+1]$ is true.

The intent is to derive a definition of OK, and associated protocol, that ensures safety and progress. Ultimately, $OK[b]$ will correspond to block b's signal set to 'clear', while $\sim OK[b]$ will correspond to block b's signal set to 'stop'.

There are two points to note about the proposed solution:

- First, if we were to insist that OK be defined as in (9), then we would need to update OK as a train moves from one block to the next, namely:

  $$\begin{aligned} t \in T \,[\!]\, (\exists b : A[t] = b \wedge OK[b+1]) \\ \rightarrow A[t] := b + 1; OK[b] := true; \\ OK[b+1] := false \end{aligned} \tag{11}$$

  (where the three statements to the right of the '$\rightarrow$' are to be executed simultaneously). I.e., as a train moves from block b to block $b+1$, set b's signal to 'clear' ($OK[b] := true$) because block b has been vacated and set $b+1$'s signal to 'stop' ($OK[b+1] := false$) because block $b+1$ has become occupied.

- Second, observe that the value of $OK[b]$ is queried *only* by a train in block $b-1$, and (as far as safety properties are concerned), it suffices that $OK[b]$ *implies* that block b be empty when queried; thus a *weaker* invariant on OK will suffice, namely:

  $$\begin{aligned} \forall b \in B : (\exists t : A[t] = b - 1) \\ \supset OK[b] \supset (\sim \exists t_1 : A[t_1] = b) \end{aligned} \tag{12}$$

  This permits $OK[b]$ to take any value when there's no train in block $b-1$.

These two points counterbalance one another; the first constrains our possible solutions, the second shows how to relax the natural, but overly-strict, initial formulation of OK. If we were willing and able to implement simultaneous updating of OK values with train movement, our problem would be solved. For a program running on a single processor, this is a fine solution. However, for distributed systems (in particular, the train system), we may wish to distribute activities across different agents, and thus need to be more tolerant of interleaving and delay (i.e., simultaneity might not be possible). Such is the case in the actual train system — station operators observe the entry of trains into their blocks and eventually change signals, but in no way are these required to be 'simultaneous' actions. This motivates our next step in the derivation.

3.5 Separating train movement from signal setting

We proceed by investigating the possibility of splitting train movement from signal setting. We have the latter done by 'operators', one positioned at each block. The activities can be represented by two (sets of) guarded commands, as follows:

$$\begin{aligned} t \in T \,[\!]\, (\exists b : A[t] = b \wedge OK[b+1]) \\ \rightarrow A[t] := b + 1; E[b+1] := true \end{aligned} \tag{13}$$

$$\begin{aligned} b \in B \,[\!]\, E[b] \rightarrow OK[b] := false; OK[b-1] := true; \\ E[b] := false \end{aligned} \tag{14}$$

$E[b]$ is used to represent the 'observation' of the entry of a train ('E' for entry) into block b. In (13), when a train moves into block $b+1$, $E[b+1]$ is simultaneously set to $true$. In (14), the operator of a block, b, responds to $E[b]$ being $true$ by simultaneously changing the signal of block, b, to 'stop' ($OK[b] := false$), the signal of the preceding block, $b-1$, to 'clear' ($OK[b-1] := true$), and discards the 'observation' of train entry ($E[b] := false$).

The danger inherent in splitting a previously atomic set of actions (11) into two (13) and (14) is that it gives rise to some new interleavings of behaviors, not all of which necessarily satisfy our requirements. To deal

with this, we take invariant (12) on OK as needing to be ensured. Following the same approach to deriving the necessary and sufficient conditions as was demonstrated in section 3.3, we find the following result for train movement:

$$t \in T \,[]\, (\exists b : A[t] = b \wedge OK[b+1] \wedge (\sim OK[b+2] \vee (\sim \exists t_2 : A[t_2] = b+2))) \rightarrow A[t] := b+1; E[b+1] := true \quad (15)$$

I.e., a train in some block b can move to the next block, $b+1$, only if $b+1$'s signal is 'clear' ($OK[b+1]$) and either $b+2$'s signal is 'stop' ($\sim OK[b+2]$) or $b+2$ is empty.
The additional conjunct
$(\sim OK[b+2] \vee (\sim \exists t_2 : A[t_2] = b+2))$
is necessary because prior to train t's movement into block $b+1$, it is possible for a train t_2 to be present in block $b+2$, but that block's signal not be set to 'stop' (which we decided was acceptable provided that no train was in the preceding block, $b+1$); however, before block $b+2$'s signal is changed to 'stop', the train t, having moved into $b+1$, might move again into $b+2$ and collide with the train t_2 still there. Fortunately, the method *derives* the necessary precondition, thus saving us the task of discovering this possibility for ourselves (this being a somewhat plausible example of something that it is a little tricky for us to think through without the aid of a formal technique).

What this suggests is that we *strengthen* the invariant (12) on OK to be:

$$\forall b \in B : (\exists t : A[t] = b-1) \supset OK[b] \supset ((\sim \exists t_1 : A[t_1] = b) \wedge ((\sim OK[b+1]) \vee (\sim \exists t_2 : A[t_2] = b+1))) \quad (16)$$

and use this new version of OK to control train movement, as before:

$$t \in T \,[]\, (\exists b : A[t] = b \wedge OK[b+1]) \rightarrow A[t] := b+1; E[b+1] := true \quad (17)$$

namely the same as equation (13), but now relying upon a strengthened definition of OK.

Repeating the derivation process by finding the preconditions on (17) that ensure satisfaction of the strengthened invariant (16) lead to addition of yet another conjunct onto the definition of $OK[b]$, requiring that block $b+2$ be empty or that at least one of the signals on blocks $b+1$ or $b+2$ to be in the 'stop' state. This process can be applied over and over again: strengthen the invariant further, do the derivation of precondition with new invariant ... This is where I apply insight to realize that the generalization of the invariant, towards which this process is iteratively converging, is the following:

$$\forall b_1, b_2 \in B : b_1 < b_2 \wedge \exists t_1, t_2 : A[t_1] = b_1 \wedge A[t_2] = b_2 \supset \exists b \in B : b_1 < b \leq b_2 \wedge \sim OK[b] \quad (18)$$

That is, the very invariant that lies at the heart of the train signalling protocol:

> *For any two trains in blocks, there is always at least one block in front of the trailing train and at or before the leading train whose signal is set to 'stop'.*

Remark: this process of iteratively calculating the guard, and generalizing if need be, corresponds to the method in [15] where weak correctness of a set of guarded commands is calculated as the limit of the chain of successive approximations, applying generalization in the case of an infinite such chain.

3.6 Added safety

Note that once we have identified the key invariant, it is easy to postulate further variations, for example, it might make for increased safety to always have at least *two* signals set to 'stop' between any two trains. This is illustrative of a beneficial side-effect of this derivational style of development — the stages that it goes through can serve as points of divergence in the space of possible designs.

3.7 Controlling OK (the signals)

The previous section 'unfolded' the invariant representing the safety condition into the guarded command expressing train movement. There remains the task of similarly unfolding it into the guarded command (14) expressing signal setting (changes to OK values).

The same formal process, making the invariant a postcondition, computing the weakest precondition, and (some) simplification, produces:

$$b \in B \,[]\, E[b] \wedge (\sim \exists t_1, t_2, b_1 : A[t_1] = b-1 \wedge A[t_2] = b_2 \wedge b_2 < b-1 \wedge (\sim \exists b_3 : b_2 < b_3 < b-1 \wedge \sim OK[b3])) \rightarrow OK[b] := false; OK[b-1] := true; E[b] := false \quad (19)$$

I.e., the operator of block b reacts when $E[b]$ holds and there's not: a train t_1 at $b-1$, a train t_2 at b_2 somewhere before $b-1$, and no intermediate signal *before* $b-1$ set to 'stop'. This complicated condition ensures that $b-1$'s signal is not the only signal set to 'stop' between a train at $b-1$ and an earlier train (since if it were the only such signal, it would not be safe to change it to 'clear').

It is possible (but not trivial) to show that there cannot be a train present in block $b-1$ while $E[b]$ holds (essentially, $E[b] = true$ arose from the movement of a train from $b-1$ into b, and until b's operator responds, there must be some signal set to 'stop' at or before $b-1$ that is preventing another train from moving into $b-1$), hence the added clause simplifies to true, and the above reduces back to (14), namely:

$$b \in B \,[]\, E[b] \rightarrow OK[b] := false; OK[b-1] := true; \\ E[b] := false$$

3.8 Ensuring Progress - a sketch

We now turn briefly to the progress goal, which is: all trains eventually advance from block to block. Stated in temporal logic terms:

$$\forall t \in T : (\exists b : A[t] = b \supset \Diamond A[t] = b + 1) \qquad (20)$$

(where $\Diamond$ is the symbol conventionally used to denote 'eventually').

Since our derived guarded command that advances trains (17) has the guard $(\exists b : A[t] = b \wedge OK[b+1])$, this implies that

$$\forall t \in T : (\exists b : A[t] = b \supset \Diamond OK[b+1]) \qquad (21)$$

It is this that drives the derivation of the necessary synchronization between station operators. As expressed so far, this means that an operator at block b must react to passage of a train (an $E[b]$ condition) *before* the operator at block $b+1$ reacts to $E[b+1]$ if the train moves on (the other way around leads to deadlock, because $b+1$'s operator's reaction is to set $OK[b]$ to *true* [which it already is], *then* b's operator sets it to *false*, after which it will never get reset to *true*!)

In fact, to get to the actual train system solution, it is necessary to make another split; in the physical system, the station operator at block b is unable to directly change the setting of the signal at block $b-1$; instead, he/she *notifies* the operator of block $b-1$, who then (eventually) reacts by changing the signal setting.

This step is very similar to the split of train movement from signal setting — what was a simultaneous activity done by a single agent becomes two activities, each done by separate agents. This necessitates reconsideration of the safety and progress goals to ensure that any new interleavings that violate either of these goals are eradicated. For brevity, I do not go through this development.

4 Related work

I have been greatly influenced by the early work of my colleagues who established the principles for, and then designed a specification language in which systems are easily specified by stating the capabilities of their components as nondeterministic choice of actions, constrained by the separately stated invariants [6].

The field of program transformation studies the derivation of implementations from specifications (e.g., [13]). This is the approach I seek to apply to distributed programs.

[7] is a representative sampling of formal work on distributed system design. In particular, [4] and [3] use weakest preconditions to justify the stepwise refinement of programs. As stated earlier, [15] gives a treatment of deriving guards on sets of guarded commands.

My study of the train example is inspired by [10], the focus of which is the issues that surround the design of composite systems — interfaces, division of tasks among multiple agents, and the like — and providing a framework in which to capture the results of human reasoning, partially assisted by mechanical reasoners. The work reported in [12] is at a similar level of concern.

5 Conclusions

My claim is that the train example illustrates how formal techniques *can* be applied in distributed system *design*. Its safety and progress goals can be succinctly stated as invariants that implicitly constrain a non-deterministic statement of the system's possible actions. The necessary and sufficient conditions on those actions can then be *derived* by a process of 'unfolding' the constraints, weakest precondition calculation, and simplification.

This particular example is interesting in that it also involves the generalization of an invariant as a conclusion of applying the derivation process iteratively. In a sense, this *discovers* the key invariant behind the

train system protocol (admittedly, I had to apply some insight to make the generalization, but it seemed a small step at that point). The need to accommodate real-world aspects such as availability of information and the division of tasks among multiple agents fits smoothly into the framework.

My derivation mixes mundane calculations (e.g., weakest precondition calculations of straight-line programs segments) that are eminently suited to mechanization, with steps that appear to require more intuitive guidance (e.g., splitting train movement apart from operator response to that movement). This suggests the opportunity to aid a human designer with a mechanical assistant capable of performing the mundane calculations automatically (and under its own initiative), and relying upon human guidance to perform the intuitive steps. Our previous investigations of system design [9] suggest that it is feasible to accumulate a set of techniques capable of *generating* the space of possible designs, but that the task of selecting among those designs requires large amounts of knowledge and insight, and hence, given the present state of the art, best left under the control of a skilled designer.

Acknowledgements

I have benefited from many discussions with Steve Fickas and his group at the University of Oregon, Eugene, on the subjects of design, distributed systems, and the like. Bob Balzer's Software Sciences Division here at ISI, of which I am a member, has for a long time studied the approach of derivation from specification, which motivates this current paper's explorations.

References

[1] G.R. Andrews. A method for solving synchronization problems. *Science of Computer Programming*, 13(1):1–21, December 1989.

[2] G.R. Andrews. *Concurrent Programming Principles and Practice.* Benjamin/Cummings, 1991.

[3] R.J.R. Back. Refinement calculus, part ii: Parallel and reactive programs. In *Stepwise Refinement of Distributed Systems*, pages 67–93. Springer-Verlag, 1990.

[4] R.J.R. Back and J. von Wright. Refinement calculus, part i: Sequential nondeterministic programs. In *Stepwise Refinement of Distributed Systems*, pages 42–66. Springer-Verlag, 1990.

[5] R. Balzer. Transformational implementation: An example. In *New Paradigms for Software Development*, pages 227–238. IEEE Computer Society Press, 1986. Originally published in IEEE TSE SE-7(1) Jan 1981 pages 3-14.

[6] R. Balzer and N. Goldman. Principles of good software specification and their implications for specification languages. In *Specification of Reliable Software*, pages 58–67. IEEE Computer Society, 1979.

[7] J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors. *Stepwise Refinement of Distributed Systems*, volume 430 of *Lecture Notes in Computer Science.* Springer-Verlag, 1990.

[8] E.W. Dijkstra. *A discipline of programming.* Prentice Hall, Englewood Cliffs, NJ, 1976.

[9] M.S. Feather, S. Fickas, and B.R. Helm. Composite system design: the good news and the bad news. In *Proceedings of the 6th Annual RADC Knowledge-Based Software Engineering (KBSE) Conference, Syracuse, NY, September 1991*, pages 16–25. IEEE Computer Society Press, 1991.

[10] S. Fickas and B.R. Helm. Knowledge representation and reasoning in the design of composite systems. *IEEE Transactions on Software Engineering*, 18(6):470–482, June 1992.

[11] J.Y. Halpern. Reasoning about knowledge: an overview. In J.Y. Halpern, editor, *Theoretical Aspects of Reasoning About Knowledge: Proceedings of the 1986 Conference, Monterey, CA*, pages 1–17. Morgan Kaufmann, 1986.

[12] J. Kramer, J. Magee, and A. Finkelstein. A constructive approach to the design of distributed systems. In *Proceedings of the 10th IEEE International Conference on Distributed Computing Systems, Paris*, 1990.

[13] B. Moeller, editor. *Proceedings of the IFIP TC2 Working Conference on Constructing Programs from Specifications, Pacific Grove, CA, USA.* North-Holland, 1991.

[14] M. Sintzoff. Eliminating bling alleys from backtrack programs. In *Automata Languages and Programming 3*, pages 531–557. Edinburgh University Press, 1976.

[15] A. van Lamsweerde and M. Sintzoff. Formal derivation of strongly correct concurrent programs. *Acta Informatica*, 12:1–31, 1979.